Building Hierarchical Grid Storage Using the Gfarm Global File System and the JuxMem Grid Data-Sharing Service

Gabriel Antoniu, Loïc Cudennec, Majd Ghareeb, Osamu Tatebe

Building Hierarchical Grid Storage Using the GFARM Global File System and the JUXMEM Grid Data-Sharing Service*

Gabriel Antoniu[1], Loïc Cudennec[1], Majd Ghareeb[1], and Osamu Tatebe[2]

[1] INRIA/IRISA, Rennes, France
[2] University of Tsukuba, Japan

Abstract. As more and more large-scale applications need to generate and process very large volumes of data, the need for adequate storage facilities is growing. It becomes crucial to efficiently and reliably store and retrieve large sets of data that may be shared at the global scale. Based on previous systems for global data sharing (global file systems, grid data-sharing services), this paper proposes a hierarchical approach for grid storage, which combines the access efficiency of RAM storage with the scalability and persistence of the global file system approach. Our proposal has been validated through a prototype that couples the GFARM file system with the JUXMEM data-sharing service. Experiments on the Grid'5000 testbed confirm the advantages of our approach.

1 Introduction

An increasing number of applications in various fields (such as genetics, nuclear physics, health, environment, cosmology, etc.) are nowadays exploiting large-scale, distributed computing infrastructures for simulation or information processing. This leads to the generation of very large volumes of data. The need to store, manage and process these data in a proper way leads to several important requirements. First, a *large storage capacity* is needed. Second, as these large volumes of data may be produced by (or used as an input for) long and costly computations, *data persistence* is essential. To address these requirements, file-based secondary storage has usually been favored in most grid storage systems. On the other hand, data need to be *efficiently* accessed in a *distributed* way at a large scale. As the cost of disk read/write operations may significantly limit the performance of data accesses, the use of faster-access RAM storage appears as a promising approach. The concept of *grid-data sharing service* [1, 2] explores this idea by providing the abstraction of a globally shared memory space, built by aggregating the RAM storage made available by thousands of grid nodes. However, the overall storage capacity is limited to the aggregated RAM storage available.

This paper proposes an architecture for large-scale, distributed grid storage whose goal is to leverage *at the same time* the efficiency of RAM accesses and the larger-capacity, persistent disk storage available on a grid. Our architecture implements a grid-scale memory hierarchy by interconnecting a grid data-sharing service (acting as a grid-scale RAM) and a grid file system.

* Corresponding author: Gabriel Antoniu, IRISA, Campus de Beaulieu, F-35042 Rennes Cedex, France. Email: Gabriel.Antoniu@inria.fr.

The remaining of the paper is organized as follows. Section 2 discusses related work. Section 3 briefly describes the two systems on which we rely: the JUXMEM grid data-sharing service and the GFARM grid file system; then, it introduces our hybrid architecture. Section 4 gives details on the interaction between the two systems. An experimental evaluation of our approach is presented in Section 5. Section 6 concludes the paper and discusses future research directions.

2 Related Work

One of the major goals of grid infrastructures is to *transparently* provide access to computational and storage resources, by hiding the details about which resources are used and where they are located, as much as possible. Regarding data storage and management, this goal is still far from being achieved, as most current grid data management systems require *explicit* data transfers before and after the computations. GridFTP [3], Chirp [4] are typical examples of two file transfer tools adapted to grid infrastructures, providing for instance support for parallel streams, authentication, checkpoint/restart in case of failures, etc. Based on such tools, catalogue-based data localization and management services have been built, such as RLS [5], Reptor [6], Optor [6], LDR [7]. Such catalogues allow the user to manually register and characterize data copies, but do not provide any support for transparent access, nor for automatic consistency maintenance.

In contrast, the concept of *grid file system* provides a familiar, file-oriented API allowing to transparently access physically distributed data through globally unique, logical file paths. The applications simply open and access such files as if they were stored on a local file system. A very large distributed storage space is thus made available to existing applications that usually use file storage, with no need for modifications. This approach has been taken by a few projects like GFARM [8], GridNFS [9], LegionFS [10], etc.

The transparent data access model is equally defended by the concept of *grid data-sharing service*, illustrated by the JUXMEM platform (described in detail in the next section). This service provides the grid applications with the abstraction of a globally shared memory in which data can be easily stored and accessed through global identifiers. Compared to the grid file system approach, this approach improves *access efficiency* by totally relying on RAM storage. Besides the fact that a RAM access is more efficient than a disk access, the system can leverage locality-optimization schemes developed within the Distributed Shared Memory (DSM) consistency protocols that serve as a basis for the system's design. However, the system's storage capacity is limited by the overall RAM available on the infrastructure.

3 Combining RAM and disk storage to achieve scalability, persistence and efficiency

As previously shown, grid file systems provide a convenient way to persistently store very large volumes of data into distributed files, whereas memory-based grid data-sharing services provide a more efficient data access. To address all these issues *at*

the same time, we propose a hierarchical storage system that combines a data sharing service with a grid file system.

3.1 The JUXMEM data sharing service

Providing a transparent data access model in an efficient way has been one of the major motivations of research efforts on distributed shared memory (DSM) systems [11].

However, the efficiency of traditional DSM consistency protocols has not proved scalable. As the grids brought forward new hypotheses (a larger scale, a dynamic infrastructure with increased failure probability), a new approach to transparent memory-based data sharing was needed. To address this challenge, the concept of *grid data sharing service* has been proposed [1]. The idea is to rely on results from several areas: location-transparent access and consistency protocols of DSM systems; algorithms for fault-tolerant distributed systems; scalability and techniques to support volatility in peer-to-peer (P2P) systems.

From the user's point of view, JUXMEM provides an API inspired by DSM systems allowing to perform memory allocation, data access through global IDs (e.g. pointers), and lock-based synchronization. Users can dynamically allocate shared memory using the juxmem_malloc primitive, which returns a global data ID. This ID can be used by other nodes in order to access existing data, through the use of the juxmem_mmap function. It is the responsibility of the implementation of the grid data-sharing service to localize the data and perform the necessary data transfers based on this ID. This is how a grid data-sharing service provides a *transparent* access to data. Both juxmem_malloc and juxmem_mmap primitives provide a local pointer that can be used to directly access the data. Note that, at the implementation level, the juxmem_mmap primitive does not rely on the mmap call. It will provide the calling client with a full copy of the data, whatever the grain of the subsequent accesses made by that client.

JUXMEM's architecture mirrors a grid consisting of a federation of distributed clusters and it is therefore expressed in terms of *hierarchical* groups of nodes. In order to cope with possible failures that may threaten data persistence, JUXMEM includes automatic replication mechanisms. When allocating a memory block, the client may specify: 1) on how many clusters the data should be replicated; 2) on how many providers in each cluster the data should be replicated; 3) the consistency protocol that should be used to manage the access to this data. As a result of the allocation procedure, a set of distributed replicas are created, called *data group*. This group has a fault-tolerant, self-organizing behavior: failures of its members are automatically detected, and failed nodes are transparently replaced, in order to maintain the replication degrees specified by the user. To do this transparently, while guaranteeing the correctness of the data accesses that may take place during the failures, we rely on fault-tolerant algorithms for group membership and atomic multicast. To favor scalability and take into account the latency hierarchy previously discussed, hierarchical adaptations of these algorithms are implemented (see [2] for details).

From an implementation point of view, the fault-tolerant algorithms used are leader-based.

The main consistency model provided by JUXMEM is *entry consistency*, first introduced in [12]. In this model, which we consider as well-adapted to grid data sharing,

processes that need to access data need to properly synchronize by acquiring a lock associated to that data. This is done by calling juxmem_acquire_read (prior to a read access) or juxmem_acquire (prior to a write access). Note that juxmem_acquire_read allows multiple readers to simultaneously access the same data. The juxmem_release primitive must be called after the access, to release the lock. These synchronization primitives allow the implementation to provide consistency guarantees according to the consistency protocol specified by the user at the allocation time.

3.2 The GFARM distributed file system

GFARM FS is a distributed file system federating local file systems. Typically, it federates local file systems on compute nodes in several clusters. A node that provides a local file system is called a *file system node*.

Physically, files can be replicated and stored on any file system node, but can be accessed transparently by a client. File replicas can be created by a GFARM *replicate* command from a file system node to another file system node. Consistency among file replicas is maintained by a close-to-open consistency like AFS [13] when a file is updated. Every other out-of-date file replicas except an up-to-date replica will be deleted when the updated file is closed.

GFARM FS is designed to achieve scalable file I/O performance in a distributed environment by *putting priority to a local file system*, and *file-affinity scheduling*. When a client is also a file system node, the local file system is a part of a GFARM FS. The data transfer from file system nodes to a client can be reduced by exploiting local file access. When a new file is created, the file is created on the local file system if there is enough disk space. When one of file replicas is stored on a local file system, the local file replica is chosen to be accessed. A file-affinity scheduling is a scheduling policy of a process allocation such that a file system node that has a specified file replica has priority to be scheduled. This increases a chance of a local file access.

GFARM FS consists of a file system metadata server and multiple file system nodes. A file system daemon called *gfsd* runs on every file system node. Client nodes access a GFARM FS by mounting it. It is developed in open source at http://sourceforge.net/projects/gfarm/.

3.3 Our proposal: a hybrid grid memory hierarchy

In order to take advantage of the more persistent, larger-capacity storage provided by the GFARM grid file system, while keeping data access efficient (i.e. not impacted by disk access delays), our approach consists in *using* GFARM *as secondary storage for the* JUXMEM *data sharing service*. The main idea is to allow applications to use JUXMEM's more efficient memory-oriented API, while letting JUXMEM to persistently store data on disk files by making calls to GFARM in the background. These calls are internally issued by JUXMEM, so they are totally transparent to the user, as illustrated on Figure 1. Besides, as explained in Section 4, their cost is also generally transparent, as the disk accesses are performed asynchronously in most cases, in order to avoid them to impact the efficiency of the application's data accesses.

With respect to persistence, as seen in the previous section, JUXMEM already enhances data availability in the presence of failures by using data replication strategies. However, every piece of data is stored in physical memory, making this system prone to hard failures of nodes. JUXMEM's consensus algorithm used to implement atomic multicast and group membership operations, supports multiple simultaneous failures within each any group of replicas, as long as a majority of nodes remain correct in the group. However, in a grid, failures of whole clusters may happen (e.g. due to air conditioning problems). This leads to the loss of all data stored exclusively in RAM, if all replicas are stored within the failing cluster. In the context of heavy, long-duration scientific applications, it may be costly (if ever possible) to regenerate the data, e.g. by restarting the computation that produced it. A more efficient approach to ensure data persistence is to use secondary, disk-based storage. Another scenario where disk storage is more appropriate than RAM storage corresponds to situation where data is read and processed a long time after it has been produced (e.g. after several weeks).

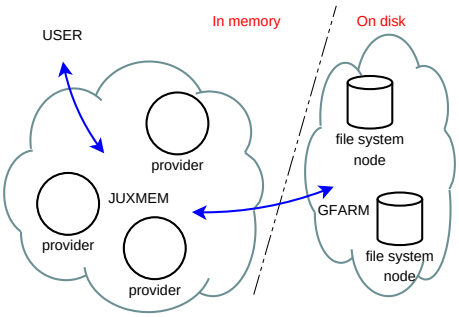

Fig. 1. The JUXMEM- GFARM architecture.

Finally, thanks to the use of GFARM, the storage space made available to applications that use JUXMEM's memory-oriented API is significantly increased: JUXMEM can act as a shared cache for actively accessed data, while GFARM ensures a large capacity for long term storage.

4 Implementing the JUXMEM-GFARM interaction

In the approach presented here, we have chosen to exhibit the memory-oriented API provided by the JUXMEM data-sharing to the grid applications: the user can dynamically allocate memory in the grid storage space, map it to its own address space and access it through local pointers. On the other side, we internally use GFARM's global IDs (i.e. globally shared file names) to persistently store JUXMEM's shared data on physical files, but these IDs, as well as the usage of this file-oriented API are hidden from the user.

We therefore need to define which JUXMEM entities should interact with GFARM, when JUXMEM should flush data to GFARM and when this data should be restored to JUXMEM. In the first version of our hybrid architecture, we decide that, for each data, a single JUXMEM provider should interact with GFARM, as a client: the data group leader. Note that JUXMEM and GFARM storage systems may share (or not!) the same physical nodes for data storage. Setting up physically distinct topologies may however be justified by the different requirements of JUXMEM (size of physical memory) and GFARM (available disk space), but also by the need to enhance fault tolerance.

4.1 Flushing data from JUXMEM to GFARM

For the sake of simplicity, we have chosen to flush data to GFARM whenever a client updates the corresponding JUXMEM memory providers at the end of a critical section during which the data is modified. According to the *entry consistency* protocol currently implemented within JUXMEM, this happens each time a client releases the lock associated to the data. Using this particular moment enforces the atomicity of the write operation in GFARM: no JUXMEM client can access the data until both shared memory and file system versions are synchronized. (This basic setting can further be refined by tuning the flush frequency as a more complex function of the data modification frequency.)

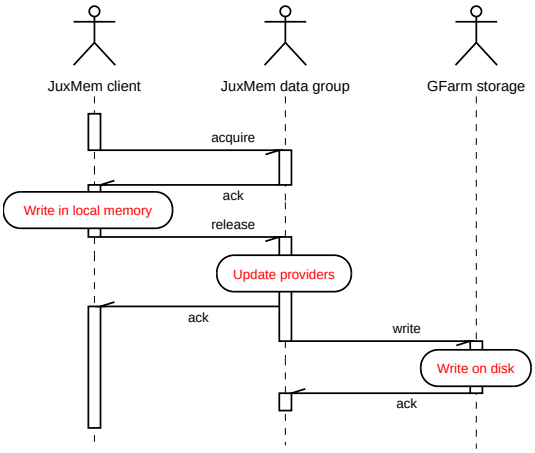

Fig. 2. Sequence diagram of basic JUXMEM-GFARM interactions.

Figure 2 describes the interactions needed between the main entities involved in a flush operation, for a basic scenario where the user performs an *acquire-write-release* sequence. Once the user acquires the lock, it gets the data from the corresponding data group; it can then modify the data in its local memory, as exclusive access is guaranteed. When the client releases the lock, the modified data is sent to the data group. All members of the data group update their copy and an acknowledgment is sent to the client to let it continue his computation, while the data group leader flushes the data into a logical GFARM file uniquely identified by a file name identical to the corresponding JuxMem data id. To further improve fault tolerance, this file containing raw data may subsequently be replicated on several GFARM file storage nodes using the dedicated GFARM *replicate* command. This GFARM replication degree can be specified when allocating the memory in JUXMEM for that data.

Note that the acknowledgement to the client is sent before the completion of the flush to GFARM. This asynchronous strategy improves performance on the client side, as it does not need to wait the data to be written in GFARM before proceeding to the next computation. However, in order to maintain data consistency, the associated lock can not be re-assigned until the data has been written in both systems. Any node which subsequently tries to acquire the associated lock in exclusive mode will be blocked until the data flush to GFARM is complete. However, read operations can proceed in parallel with the flush operation. The benefits of our asynchronous scheme are therefore real for write-once data, or if the frequency of writes to the same data does not fall under a

threshold where clients have to wait for JUXMEM and GFARM updates. This threshold depends on the data size and on network communication performance.

Data flushing from JUXMEM to GFARM introduces an overhead discussed in section 5. If during a computation step the write frequency to some data is too high, the time needed to flush into GFARM may degrade the user application performances. In that case, it may be assumed that this computation step is short enough to decide a lower frequency for data flushes to GFARM (e.g. one flush every ten modifications instead of one flush after *each* modification). Flush frequency to GFARM should thus be seen as a finely tuned parameter that makes a trade-off between reliability and performance.

4.2 Restoring data from GFARM to JUXMEM

Restoring data consists in reading a data from the file system in order to store it in the data sharing service. This operation occurs in two cases: 1) a JUXMEM client accesses a data that is no longer in the service, because of failures or because it was produced a long time before; and 2) the user would like to restore a given version of the data within a rollback procedure. As for flushing, one of the simplest way to achieve the restoring operation is to use the data group leader for reading the data from the file system.

When a client requires access to some data, the JUXMEM service is queried to check if the corresponding data group is still present in memory. If not, the corresponding file (if any) is searched for in the GFARM file system, using the *data ID* specified by the client. If the data is found in GFARM, a corresponding JUXMEM data group is created, then the data is sent to the client.

5 Feasibility study: evaluation

We have designed and implemented a prototype in which the JUXMEM data-sharing service uses GFARM according to the interaction scheme explained in Section 4. We use our prototype to run a synthetic application simulating producer-consumer access patterns. Each piece of data is written *once* and can be accessed independently. This scenario is inspired by real applications, such as climate (e.g. ocean-atmosphere) modeling applications based on code-coupling [14]. We measure the average time to write (respectively read) a piece of data by performing 20 successive accesses. *The goal of this preliminary evaluation is to show that thanks to our combined approach, the* JUXMEM *service is enhanced with persistence guarantees provided by* GFARM*, whereas the access cost remains the same (i.e. not impacted by disk access delays) in most cases.*

Evaluations are performed using the Grid'5000 [15] testbed. We use 7 nodes of a Grid'5000 cluster made of Intel Xeon 5148 LV CPUs running at 2.3 GHz, outfitted with 4 GB of RAM each, SATA hard drives (57 MB/s peak throughput) and interconnected by a Gigabit Ethernet network. The theoretical maximum network bandwidth is thus 125 MB/s; however, if we consider the IP and TCP header overhead, this maximum becomes slightly lower: 117.5 MB/s when MTU = 1500 B. GFARM runs on 3 nodes (a metadata server, a cache metadata server, a file system node) and JUXMEM uses 4 nodes (a producer, a consumer, a manager and a memory provider also acting as a GFARM client).

In order to have a reference for our evaluation, we first ran the scenario described above using GFARM only, without JUXMEM. Read and write access times for GFARM for different data sizes, are provided on Figures 3(a) and 3(b). The average read throughput is 69 MB/s and the average write throughput is 42 MB/s. Note that GFARM's read throughput is higher than the peak hard drive throughput (57 MB/s), which indicates that GFARM benefits here from some cache effects. These results are given as reference values for comparison with the access times provided by our hierarchical grid storage system. To evaluate the cost of writing data using our hierarchical grid storage, we consider two scenarios: a common-case scenario and a worst-case scenario.

In the first scenario, we consider that write accesses to a same piece of data are infrequent. In such a case, thanks to our asynchronous scheme, the data flush to GFARM after a write session is complete before the next write to that data into JUXMEM. This is the case of all write accesses for our producer-consumer scenario, because all data are written only once. Figure 3(b) indicates a write throughput of 89 MB/s for our hierarchical storage system, *equal to the pure* JUXMEM *throughput*. This represents an improvement by 112% compared to the pure GFARM throughput (42 MB/s), essentially due to the relative costs of memory accesses compared to disk accesses. Although the data is written to GFARM in both schemes, the improvement is possible thanks to the fact that, once the data is written into JUXMEM, the producer can continue its computation while the data is flushed into GFARM. Note that the JUXMEM write cost includes the cost of synchronization (acquire/release).

In a second scenario, we consider that write accesses to a same piece of data are frequent. In this case, each write (but the first one) has to wait for the previous one to be flushed into GFARM. Basically, the access time is therefore the time to write into JUXMEM plus the time to write into GFARM. This leads to a 26 MB/s write throughput, which is slightly (7%) lower than the theoretical throughput (28.5 MB) that can be estimated based on the separate write throughputs of JUXMEM and GFARM. In this case, as explained at the end of Section 4.1, it may be reasonable to reduce the flush frequency, i.e. to allow several write sessions to JUXMEM to proceed before flushing data to GFARM.

We equally analyze the cost of reading accesses for our hierarchical grid storage using two scenarios. We first consider that the data is present in JUXMEM (i.e. in physical memory). Figure 3(a) shows the time to read a data in such a configuration. The read throughput for our hierarchical prototype is again *equal to* JUXMEM*'s read throughput* and reaches 100 MB/s in this case, i.e. an improvement by 45% compared to an optimized GFARM throughput of 69 MB/s (which already benefits from some cache effects, as explained above). For reference, we remind the reader that, theoretically, the maximum network throughput is 117 MB/s. In this configuration, we can claim that our hierarchical grid storage provides GFARM's persistence guarantees, while the user only "pays" the cost of JUXMEM accesses (which includes, as previously, the synchronization cost, and does not benefit of any specific optimization).

We also consider a reading scenario where the data is not hosted by any JUXMEM provider in physical memory, but remains available in the GFARM file system. The time to read includes the time to retrieve the data from GFARM, to store the data in JUXMEM and to send it to the client. In this configuration, the throughput reaches 39 MB/s. This

is a worst-case scenario that only occurs when the data is not present in JUXMEM. Note that this cost is incurred only once, as subsequent read accesses to the same piece of data benefit from JUXMEM's access cost (100 MB/s).

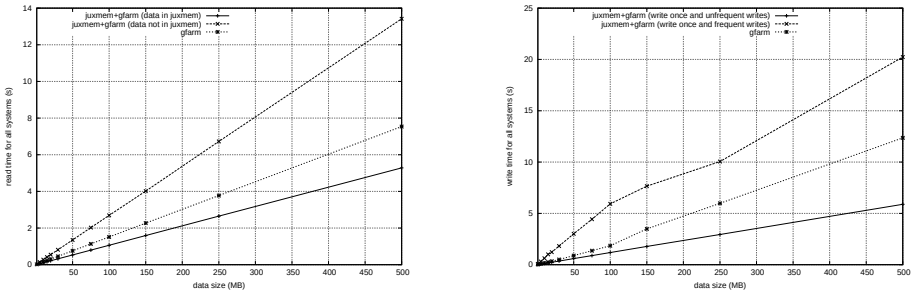

Fig. 3. Cost of read (a) and write (b) operations in JUXMEM and GFARM.

6 Conclusion and future work

While grid file systems provide an elegant solution for *persistent* storage of *large volumes of data* on physically distributed files, the concept of grid data-sharing service offers *efficient* access to globally shared data by relying on RAM storage. We propose a hierarchical grid storage system that simultaneously addresses these issues, based on the JUXMEM *grid-data sharing service* and on the GFARM *grid file system*. The main idea is to allow applications to use JUXMEM's efficient memory-oriented API, while letting JUXMEM persistently and transparently store data on GFARM disk files.

Our experiments performed on the Grid'5000 testbed confirm the advantages of our approach: in most cases the data access cost is not impacted by the cost of disk accesses (100 MB/s read throughput, 89 MB/s write throughput). At the same time, thanks to GFARM, the storage space made available to applications that use JUXMEM's memory-oriented API is significantly increased: JUXMEM can act as a shared cache for actively accessed data, while GFARM ensures a large capacity for long term, persistent storage.

As a future work, we plan to extend our experiments to more complex configurations where replication is used to improve fault tolerance. We plan to evaluate the induced overhead, and to address this issue using parallel data accesses between JUXMEM providers and GFARM storage nodes. We equally intend to develop and compare multiple cache strategies allowing JUXMEM to efficiently act as a cache for actively used data, while GFARM would serve for long-term storage.

Acknowledgment This work has been supported by the Sakura programme for bilateral Japan-France collaborations, by the AIST (Tsukuba, Japan), by the French National

Agency for Research project LEGO (ANR-05-CIGC-11) and by the NEGST France-Japan research collaboration programme. It has been also supported by a grant of Sun Microsystems and a grant from the Regional Council of Brittany, France.

The experiments presented in this paper were carried out using the Grid'5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see http://www.grid5000.fr/).

References

1. Antoniu, G., Bougé, L., Jan, M.: JuxMem: An adaptive supportive platform for data sharing on the grid. Scalable Computing: Practice and Experience **6**(3) (November 2005) 45–55
2. Antoniu, G., Deverge, J.F., Monnet, S.: How to bring together fault tolerance and data consistency to enable grid data sharing. Concurrency and Computation: Practice and Experience **18**(13) (November 2006) 1705–1723
3. Allcock, B., Bester, J., Bresnahan, J., Chervenak, A.L., Foster, I., Kesselman, C., Meder, S., Nefedova, V., Quesnel, D., Tuecke, S.: Data management and transfer in high-performance computational grid environments. Parallel Comput. **28**(5) (2002) 749–771
4. Chirp protocol specification. Available at http://www.cs.wisc.edu/condor/chirp/
5. Dunno, F., Gaido, L., Gishelli, A., Prelz, F., Sgaravatoo, M.: DataGrid prototype 1. EU-DataGrid collaboration. In: Proc. of the TERENA Networking Conf., Limerick, Ireland (June 2002)
6. Kunszt, P.Z., Laure, E., Stockinger, H., Stockinger, K.: File-based replica management. Future Generation Computing Systems **21**(1) (2005) 115–123
7. Lightweight data replicator. Available at http://www.lsc-group.phys.uwm.edu/LDR/
8. Tatebe, O., Morita, Y., Matsuoka, S., Soda, N., Sekiguchi, S.: Grid datafarm architecture for petascale data intensive computing. In: Proc. 2nd IEEE/ACM Intl. Symp. on Cluster Computing and the Grid (Cluster 2002), Washington DC, USA, IEEE Computer Society (2002) 102
9. Honeyman, P., Adamson, W.A., McKee, S.: GridNFS: global storage for global collaborations. In: Proc. IEEE Intl. Symp. Global Data Interoperability - Challenges and Technologies, Sardinia, Italy, IEEE Computer Society (June 2005) 111–115
10. White, B.S., Walker, M., Humphrey, M., Grimshaw, A.S.: LegionFS: a secure and scalable file system supporting cross-domain high-performance applications. In: Proc. 2001 ACM/IEEE Conf. on Supercomputing (SC '01), New York, NY, USA, ACM Press (2001) 59–59
11. Protic, J., Tomasevic, M., Milutinovic, V.: Distributed shared memory: concepts and systems. IEEE Paralel and Distributed Technology **4**(2) (1996) 63–71
12. Bershad, B.N., Zekauskas, M.J., Sawdon, W.A.: The Midway distributed shared memory system. In: Proc. 38th IEEE Intl. Computer Conf. (COMPCON Spring '93), Los Alamitos, CA (February 1993) 528–537
13. Kazar, M.L.: Synchronization and caching issues in the andrew file system. In: USENIX Winter. (1988) 27–36
14. Valcke, S., Caubel, A., Vogelsang, R., Declat, D.: OASIS 3 user's guide. Technical Report TR/CMGC/04/68, CERFACS, Toulouse, France (2004)
15. The Grid'5000 project. Available at http://www.grid5000.org/